Docket No.: DE920000101US1

Inventor: Friedemann Baitinger et al.

For: **METHOD AND SYSTEM FOR EFFICIENT ACCESS TO REMOTE I/O FUNCTIONS IN EMBEDDED CONTROL ENVIRONMENTS**

APPLICATION FOR UNITED STATES

LETTERS PATENT

"Express Mail" Mailing Label No. ER363647528US
Date of Deposit: November 14, 2003

I hereby certify that this paper is being deposited with the United States Postal Service as "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to: Box Patent Application, Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450.

Name: Susan L. Nelson

Signature: Susan L. Nelson

INTERNATIONAL BUSINESS MACHINES CORPORATION

METHOD AND SYSTEM FOR EFFICIENT ACCESS TO REMOTE I/O FUNCTIONS IN EMBEDDED CONTROL ENVIRONMENTS

[0001] The present application is a continuation of International Patent Application PCT/EP02/04837 filed March 3, 2002 as a PCT application.

Field of the Invention:

[0002] The present invention relates, in general, to accessing I/O functions. More specifically, the invention relates to accessing I/O functions that are remotely attached to a microcontroller. Still more specifically, the present invention deals with accessing such remotely attached I/O functions in embedded control environments.

Background of the Invention:

[0003] In distributed embedded control environments I/O engines are often remotely attached to the embedded controller through a serial link. Typically access to these devices is facilitated by directly programming the serial link controller hardware.

[0004] Fig. 1 shows such an environment where an embedded controller needs to operate a series of I/O devices which are attached at the remote end of a link.

[0005] At the lowest level a microprocessor accesses I/O resources either through its I/O address space for architectures such as the Intel x86, or through its memory address space for architectures such as the IBM PowerPC architecture. The software entity that controls the hardware at this level is usually called

a Device Driver. Towards higher level software applications the Device Driver provides a generic view of the device. In UNIX systems, it is typically visible in the hierarchical file- system name space in the /dev tree and access to it is facilitated using regular file-system semantics and interfaces like:

- open();
- close();
- read();
- write();
- ioctl()

[0006] In embedded control applications it is not always possible to attach all I/O devices such that they can be directly addressed at the microprocessor bus level as shown in Fig. 2. The I/O functions are often implemented in a standalone ASIC which can be attached to the embedded controller's microprocessor via some link hardware. Consequently, the resources of the I/O devices are not directly visible in the I/O address space or memory address space of the microprocessor. What is visible to the microprocessor though are the resources of the link hardware that connects the I/O ASIC with the embedded controller.

[0007] Since no existing device driver can be used in this situation the full software path to the I/O devices needs to be facilitated through custom device interface software. The need for this custom device interface software is a limiting factor for time-to-market which is often key in the embedded control market nowadays.

[0008] In addition custom implementation of the device driver layer increases the probability of shipping software bugs to the field which cannot be easily recovered from. Unlike regular PC applications where it is relatively simple to download a new version of a previously buggy application over the Internet, it is not easily possible to replace faulty software in embedded control applications.

[0009] Until recently this problem did not exist because embedded control applications traditionally used custom operating systems, custom device drivers, and custom hardware. Since the problem is rather new, standard solutions are not yet available, however, with the emerging push towards standardization in embedded control environments new solutions will become more prevalent.

[0010] A major driver towards this standardization is Linux and the overall OpenSource movement.

Summary of the Invention:

[0011] It is therefore an object of the present invention to provide a method for accessing I/O devices in embedded control environments thereby greatly reducing the programming effort.

Brief Description of the Drawings

[0012] The invention will in the following be described in more detail in conjunction with the drawings, in which:

[0013] Fig. 1 schematically shows an I/O card remotely attached to a microprocessor according to the state of the art;

[0014] Fig. 2 schematically depicts a microprocessor bus attached I/O;

[0015] Fig. 3 schematically shows a hardware solution according to the invention;

[0016] Fig. 4 schematically depicts a software structure with a Device Abstraction Layer (DAL) according to the present invention; and

[0017] Fig. 5 schematically depicts a sequence of events describing the DAL operation according to the invention.

Detailed Description of the Preferred Embodiment:

[0018] The present invention allows for reuse of existing device drivers which are available for common I/O devices as long as they are mapped to the processor's I/O- or memory-address space in well known locations and with a well known layout. Reusing existing device driver code greatly improves the time-to-market capability and it reduces the software test effort as well.

[0019] It has to be mentioned that the present invention does not change the existing device drivers but adapts them to a fictitious device.

[0020] For remote but well known device types such as serial ports which are often referred to as UARTs (Universal Asynchronous Receiver/Transmitter) where device drivers exist for nearly all operating systems and Real Time Control Programs (RTOS), a Device Abstraction Layer (in the following called "DAL") is provided which maps the devices' resources into the

microprocessor I/O address space or memory address space as if the remote device was locally available to the microprocessor.

[0021] Table 1 shows an example of UART resources as they are visible to the microprocessor as registers either in the I/O address space or the memory address space. By doing so, it is possible to instantly reuse existing device drivers and all the existing application software that uses them without the need to rewrite a vast amount of complex code.

Table 1 UART Registers

Offset	R/W	Description
0	R	Receive Buffer
	W	Transmitter Holding Register
1	R/W	Interrupt Enable Register
2	R	Interrupt Identification Register
3	R/W	Line Control Register
4	R/W	Modem Control Register
5	R	Line Status Register
6	R	Modem Status Register
7		Unused
8	R/W	Divisor Latch (LSB)
9	R/W	Divisor Latch (MSB)

[0022] The implementation of the DAL is done in either one of the following ways:

1. Embedding control hardware to redirect the requests and responses over the link to the remote device and still maintain the device abstraction.

2. Using the microprocessor's memory management unit to cause a program exception as soon as the DAL's resources are accessed which is done by providing a thin layer device abstraction which executes in the context of the exception handler.

[0023] As to the hardware solution, figure 3 shows how a microcontroller 2 or equivalent processor core within an ASIC device accesses the link logic 6 by a system bus 4 via a local register set 8 combined with a state machine. The link logic 6 may communicate with a respective link logic 10 of the remote device, e.g., by using an appropriate and suitable protocol via link 16. The link logic 10 connects to a remote register set 12 containing a second state machine and attaches to the remote device 14 via a bus 18. It has to be mentioned that the link logic 6 can be any kind of a data exchange (data link) between two stations A and B. This may be achieved by a simple serial interface, a LAN, a private network or a global network, etc., but is not restricted thereto.

[0024] The microcontroller 2 triggers, e.g., a read or write operation to the remote device 14 by writing to the local register set 8. This in turn activates the state machine in 8 and controls the transfer of data to the link logic 6. The link logic units 6 and 10 transfer the request posted in 8 to the register set 12 and subsequently activate the respective state machine to

execute the requested operation in 14, i.e., to write or read any data to/from the remote device 14.

[0025] After completion of the operation the state machine 12 triggers a response to be sent back via the link 16 and stored in a register of unit 8. This action may also cause a notification to the micro controller, e.g., an interrupt. Units 2, 6 and 8 may be designed as one single ASIC device integrating all functions or by use of standard vendor components that are interconnected as discrete modules.

[0026] The same approach is valid for the units 10, 12 and 14. While the link hardware 6, 10, 12 and 16 is required to overcome the physical distance between units 2 and 14, unit 8 may translate partially or fully to a software layer below the DAL. This allows for cost tradeoffs between hardware and software.

[0027] Solution 1 mentioned above is used for rather simple I/O controllers where the hardware design effort is affordable and where performance requirements dictate it.

[0028] Solution 2 is a software solution and is used where the hardware cost cannot be afforded but one can live with the overhead of entering and leaving the exception handler context. This solution will be described in more detail in the following.

[0029] Fig. 4 schematically depicts a software structure with a Device Abstraction Layer (DAL) according to the present invention.

[0030] The general mechanism used by the DAL implementation is a means for generating an exception during instruction execution upon accessing a virtual resource that does not physically exist.

[0031] Two methods may be used to trigger the DAL, i.e.,

1) Use of the microprocessors' management unit's capability to generate an exception upon accessing a memory location where no real memory is mapped to; and

2) Use of the microprocessors' execution unit's capability to generate an exception upon execution of a privileged instruction.

[0032] The following description of the DAL operation is based on method 1 but is also applicable to methods 2 and 3.

[0033] The Device Abstraction Layer "DAL" facilitates the virtualization of the remotely attached device and makes it appear to device driver software as if it was locally attached to the microprocessor bus.

[0034] The following sequence of events as shown in Fig. 5 describes the operation of the DAL according to the described software solution and shows how it interacts with the rest of the system:

[0035] During steady state operation, at some point in time the device driver for the virtual device may want to issue an I/O operation to the remote device. The device driver writes to the appropriate memory area in the microprocessor's memory address range (Step 1).

[0036] The Memory Management Unit of the microprocessor has been programmed such that read/write accesses to the relevant address range cause a Page-Fault Exception (Step 2).

[0037] Subsequently, the Page-Fault Exception Handler is invoked and a decision is made whether the exception needs to be handled by the DAL or treated like a regular page-fault.

[0038] The Exception Handler transfers control to the DAL (Step 3). It provides sufficient information for the DAL to decode the operation that the device driver actually wanted to perform with the device.

[0039] The DAL updates its internal state machine accordingly and issues the Start-I/O operations (Step 4) to the real device over the Link Driver as needed.

[0040] Once the I/O operation is complete at the remote device, an I/O complete interrupt is signaled to the microprocessor (Step 5).

[0041] Now, the interrupt handler decodes the interrupt source and passes control to the I/O complete handler in the Link Driver (Step 6).

[0042] The I/O complete handler decodes the event and passes DAL related events back to the DAL (Step 7).

[0043] The DAL retrieves the I/O information from the device via the Link Driver, updates its internal finite state machine tables and signals an I/O complete event to the device driver (Step 8).

[0044] The device driver issues read/write operations to the memory map of the device in order to obtain status of the I/O operation. Access to these memory locations again causes page-fault exceptions which are fed to the DAL which knows the

relevant information from the I/O device already and thus can make it available to the device driver by completing the interrupted memory read/write operations.

[0045] While the preferred embodiment of the invention has been illustrated and described herein, it is to be understood that the invention is not limited to the precise construction herein disclosed, and the right is reserved to all changes and modifications coming within the scope of the invention as defined in the appended claims.